LLMalMorph：论使用大型语言模型生成恶意软件变体的可行性

摘要

Large Language Models (LLMs) have transformed software development and automated code generation. Motivated by these advancements, this paper explores the feasibility of LLMs in modifying malware source code to generate variants. We introduce LLMalMorph, a semi-automated framework that leverages semantical and syntactical code comprehension by LLMs to generate new malware variants. LLMalMorph extracts function level information from the malware source code and employs custom-engineered prompts coupled with strategically defined code transformations to guide the LLM in generating variants without resource-intensive fine-tuning. To evaluate LLMalMorph, we collected 10 diverse Windows malware samples of varying types, complexity and functionality and generated 618 variants. Our thorough experiments demonstrate that it is possible to reduce the detection rates of antivirus engines of these malware variants to some extent while preserving malware functionalities. In addition, despite not optimizing against any Machine Learning(ML)-based malware detectors, several variants also achieved notable attack success rates against an ML-based malware classifier. We also discuss the limitations of current LLM capabilities in generating malware variants from source code and assess where this emerging technology stands in the broader context of malware variant generation.

关键词：

Abstract

大型语言模型（LLMs）已经改变了软件开发和自动化代码生成。受这些进展的激励，本文探索了 LLMs 修改恶意软件源代码以生成变种的可行性。我们引入了 LLMalMorph，这是一个半自动化框架，它利用 LLMs 对代码的语义和语法理解来生成新的恶意软件变种。LLMalMorph 从恶意软件源代码中提取函数级信息，并采用定制设计的提示词结合策略性定义的代码转换，来指导 LLM 生成变种，而无需资源密集型的微调。为了评估 LLMalMorph，我们收集了 10 个不同类型、复杂度和功能的多样化 Windows 恶意软件样本，并生成了 618 个变种。我们详尽的实验表明，在保持恶意软件功能的同时，可以在一定程度上降低这些恶意软件变种对防病毒引擎的检测率。此外，尽管没有针对任何基于机器学习（ML）的恶意软件检测器进行优化，一些变种在对抗一个基于 ML 的恶意软件分类器时也取得了显著的攻击成功率。我们还讨论了当前 LLM 在从源代码生成恶意软件变种方面的能力局限性，并评估了这项新兴技术在更广泛的恶意软件变种生成背景下的现状。

Key Words:

目录

插图

表格

主要符号对照表

LLM	大语言模型的英文缩写

第 1 章 引言

恶意软件（Malware）继续随着技术的快速扩张而激增。到 2025 年，网络犯罪造成的损失预计将达到每年 10.5 万亿美元 [1]。每秒大约发生 19 万起新的恶意软件事件 [2]，而 2024 年勒索软件的平均赎金要求预计将达到每次攻击 273 万美元，较往年急剧上升 [3]。尽管经过数十年的研究和缓解努力，这些数字突显了恶意软件研究在当今不断演变的威胁环境中的紧迫重要性。

现代最具变革性的 AI 技术之一是大型语言模型（LLMs），它在自然语言处理（NLP）[4]–[6]、代码生成 [7]–[12] 以及代码编辑和重构等软件工程任务 [13]–[15] 中展现了非凡的能力。鉴于这些优势和进步，利用 LLMs 进行恶意软件源代码转换是自然的发展。最近一项针对全球行业 1800 名安全负责人的调查 [16] 发现，74% 的人正经历着显著的 AI 驱动的威胁，60% 的人感觉准备不足，无法抵御这些威胁。尽管当前的模型仅从文本生成功能完整的恶意软件存在显著局限性，但研究表明它们可以生成恶意行为者能够组装成可操作恶意软件的代码片段 [17]。LLM 能力的进步与恶意软件威胁的演变相结合，为对手使用这些模型创建新恶意软件并将现有代码库变异成更难以捉摸和更具破坏性的变种铺平了道路。尽管恶意软件源代码比二进制文件更难获取，但能够访问源代码的对手，例如恶意软件作者、泄露存储库的用户或修改开源恶意软件的人，仍然可以利用 LLMs 生成新的、更难检测的变种。这些模型使攻击者能够持续精进和扩展其武器库，从而大规模增加恶意活动的持久性和规避性。

1.1 先前的研究

先前的研究提出了各种创建恶意软件变种的方法 [17]–[23]。然而，这些方法在至少以下一个方面表现出局限性（如表1.1所示）(A) 大多数现有方法没有利用 LLMs 来转换恶意软件的源代码 [18]–[23]；(B) 大多数方法依赖迭代算法来生成恶意软件变种 [18]–[20], [22], [23]；(C) 使用 LLMs 进行变种生成的方法，直接从成功率低的提示词开始 [17]。此外，尚不清楚生成的恶意软件在规避广泛使用的防病毒引擎方面是否表现更优。鉴于目前的情况，我们的工作引入了一种与现有恶意软件变种生成方法截然不同的方法。与大多数先前主要依赖基于对抗性机器学习或基于搜索的方法的研究不同，我们的方法独特地利用 LLMs 在源代码级别进行操作。基本上，它从恶意软件源

代码开始，以高成功率和最少的手动工作生成变种。此外，我们的方法不需要迭代训练或基于搜索的优化，这使其与现有的恶意软件转换方法根本不同。因此，我们提出了一个尚未充分探索的新研究方向。

表 1.1　与先前研究的对比

方法	源代码	LLM 使用	无需训练或迭代	逃逸提升
Qiao, Yanchen, et al. [18]	否	否	否	是
Tarallo [23]	否	否	否	是
Malware Makeover [20]	否	否	否	是
MalGuise [22]	否	否	否	是
Ming, Jiang, et al. [21]	否	否	是	是
AMVG [19]	是	否	否	是
Botacin et al. [17]	否	是	是	否
LLMalMorph	是	是	是	是

1.2 问题描述

鉴于现有方法的局限性以及 LLMs（特别是代码生成方面）的最新进展，我们旨在回答以下问题——我们能否利用预训练 LLMs 的生成能力，无需额外微调，来开发一个半自动化且高效的框架，以生成保留功能语义的恶意软件变种，这些变种能够规避广泛使用的防病毒引擎和机器学习分类器？

1.3 我们的方法

在本文中，我们对上述问题给出了肯定的回答。我们设计、实现并评估了 LLMalMorph——一个专门用于生成用 C/C++ 编写的 Windows 恶意软件功能变种的框架。我们只专注于 Windows 恶意软件，因为它在消费者和企业环境中广泛使用，仍然是恶意软件最常针对的操作系统 [24], [25]。

LLMalMorph 结合了自动化代码转换和人工监督来生成恶意软件变种。该框架利用一个开源的 LLM，应用精心设计的转换策略和提示词工程，在保持结构和功能完整性的同时，高效地修改恶意软件组件。人机协同（human-in-the-loop）过程处理复杂转换和多文件恶意软件中的错误，允许进行调试和配置调整。这种半自动化方法也

使我们能够量化基于 LLM 从源代码生成恶意软件变种中的人力投入。

1.4 实验和分析

我们选择了 10 个不同复杂度的恶意软件样本，使用 6 种代码转换策略结合一个 LLM 生成了 618 个变种。我们使用主要依赖基于签名的检测和静态分析的引擎的 VirusTotal[1]和 Hybrid Analysis[2]评估了防病毒（AV）检测率，并测试了语义保留性。代码优化（Code Optimization）策略在两种工具上均持续实现了较低的检测率。平均而言，相对于每个样本的基准检测率，LLMalMorph 在 VirusTotal 上将简单样本的检测率降低了 31%，将三个更复杂样本的检测率降低了 10% 至 15%；在 Hybrid Analysis 上，与各自基准相比，四个样本的检测率降低了 8% 至 13%。除了 AV 工具外，我们还在一个基于机器学习（ML）的恶意软件分类器上评估了 LLMalMorph，并观察到在特定样本上，优化（Optimization）策略和安全（Security）策略取得了较高的攻击成功率（分别高达 89% 和 91%）。诸如优化、安全和 Windows API 修改等策略需要更多手动编辑，其中 Windows 和安全策略需要更高的调试投入。值得注意的是，四个样本中超过 66% 的规避型变种保留了其语义，这证明了 LLMalMorph 生成功能规避型恶意软件的能力。

1.5 贡献

总结而言，我们有以下贡献：

• 我们设计并实现了 LLMalMorph，一个实用的 Windows 恶意软件变种生成框架，它使用一个开源的 LLM 和基于提示词（prompt-based）的代码转换。

• 我们在 LLMalMorph 中设计了一个人机协同（human-in-the-loop）机制，以解决 LLM 在调试多文件恶意软件源代码和项目级配置方面的局限性。

• 我们进行了广泛的实验，从 10 个样本生成了 618 个恶意软件变种，并评估了它们在 VirusTotal 和 Hybrid Analysis 上的检测率和语义保留性，以及在一个机器学习分类器（ML Classifier）上的攻击成功率。

• 我们使用代码编辑工作量（code editing workload）比较了不同代码转换策略的有效性，并讨论了 LLM 所犯错误的类型。

[1]https://www.VirusTotal.com/gui/home

[2]https://hybrid-analysis.com/

1.6 开源

LLMalMorph 框架及其所有相关组件可在 Github[3]找到。

[3]https://github.com/AJAkil/LLMalMorph

第 2 章 背景

本节描述与恶意软件（malware）、其检测系统以及大型语言模型（LLMs）相关的各种预备知识。

2.1 恶意软件和检测手段

恶意软件（Malware）指的是对手或攻击者用来在用户不知情的情况下，未经授权访问数字设备以破坏或窃取敏感信息的恶意程序 [26]。它是一个统称（umbrella term），用于描述广泛的威胁，包括木马（Trojans）、后门（backdoors）、病毒（viruses）、勒索软件（ransomware）、间谍软件（spyware）和僵尸程序（bots）[27]，针对多种操作系统，如 Windows、macOS、Linux 和 Android，以及各种文件格式，如可移植可执行文件（Portable Executable, PE）、MachO、ELF、APK 和 PDF [28]。在入侵系统后，恶意软件可以执行各种恶意活动，例如渗透网络、加密数据以勒索赎金或降低系统性能。

检测引擎和工具采用各种方法和工具来检测恶意软件。它们可以大致分为静态（static）、动态（dynamic）和混合（hybrid）方法 [28], [29]。静态检测在不执行恶意软件的情况下对其进行分析，依赖于诸如 PE 头信息（PE header information）、可读字符串（readable strings）和字节序列（byte sequences）等特征 [28]。动态检测涉及在受控环境（例如沙箱，sandboxes）中执行恶意软件，以监视运行时行为，如注册表修改（registry modifications）、进程创建（process creation）和网络活动（network activity）[28], [29]。混合检测结合了静态和动态特征，使用诸如操作码（opcodes）、对系统的 API 调用（API calls to the system）和控制流图（control flow graphs, CFGs）等数据 [28]。此外，基于启发式的检测（heuristic-based detection）使用启发式规则（heuristic rules）静态分析代码并动态分析行为，以确定恶意性 [30]。

2.1.1 LLMs 和提示词工程

LLMs 通过在翻译、摘要等任务中的卓越表现，改变了自然语言处理（NLP）的格局。基于 transformer 架构 [31]，LLMs 利用了自注意力机制（self-attention mechanisms）。它们以自监督（self-supervised）方式在大规模语料库上进行预训练，以形成对语料库的深度上下文理解。预训练后，这些模型经过微调（fine-tuned）或指令微调（instruction-tuned）以执行特定任务。

LLMs 在编程任务中也展现了显著的能力，一些专门模型在大量代码和自然语言指令上进行了训练 [8], [9], [11], [12], [14]。这些模型最突出的特性之一是在推理过程中无需任务特定微调即可生成零样本代码（zero-shot code）（无需显式示例或参考）。这是通过提示词工程（prompt engineering）实现的，其中精心设计的输入提示词（prompts）指导模型生成期望的输出 [32]，使其成为代码合成（code synthesis）和重构（refactoring）等编程活动的多功能工具。

第 3 章 概述

在本节中，我们正式定义我们的问题，并阐述这些挑战及相应的解决方案。

3.1 A. 问题描述

令 M 表示一个由 F 个文件组成的恶意软件程序，其中第 i 个文件 (1≤i≤F) 包含 G 个函数，记作 $\{f_1^i, f_2^i, ..., f_G^i\}$。对于由语言模型（LLM）应用的给定转换策略 s，我们的目标是生成一个恶意软件变种 M_s，其中第 i 个文件包含使用策略 s 生成的修改后函数 $\{\hat{f}_1^i, \hat{f}_2^i, ..., \hat{f}_j^i\}$，同时保留未修改的函数 $\{f_{j+1}^i, ..., f_G^i\}$。该过程首先涉及从第 i 个文件中提取第 j 个函数 f_j^i，并构建一个提示符 $p_s||f_j^i$，该提示符包含转换策略 s、提取的函数 f_j^i 以及相关上下文信息（如全局变量和头文件）。然后我们得到转换后的函数 $\hat{f}_j^i = LLM(p_s||f_j^i)$ 。随后，将修改后的函数 f_j^i 合并回源代码文件 i，生成一个修改后的文件，其中函数 $\{\hat{f}_2^i, ..., \hat{f}_j^i\}$ 被修改，而其余函数 $\{f_{j+1}^i, ..., f_G^i\}$ 保持不变。最后，重构的文件被编译以生成变种恶意软件 $\hat{M}_s$。

3.2 B. 挑战与解决方案概述

我们现在讨论指导我们设计 LLMalMorph 的主要挑战和解决方法。

3.2.1 (C1) 编辑恶意软件源码中的上下文与结构挑战

由于在训练语料库中包含了大量开源代码库 [8]–[10], [12]，LLM 研究的最新进展极大地改进了跨多种语言的代码生成。除了生成代码，使用 LLM 进行代码编辑和重构也正受到关注 [13], [14]。一个关键挑战是上下文限制：提供完整的源代码和转换指令通常会超出模型的输入能力并阻碍指令遵循，特别是对于较小的模型。此外，恶意软件代码中的功能通常分布在多个文件中，这进一步使编辑过程复杂化。在处理 C 和 C++ 时，这个问题变得更加明显，因为它们经常导致产生无法编译或无法按预期执行的错误代码 [33]。这与恶意软件高度相关，因为这些程序经常利用系统调用 API 进行注册表修改、执行网络系统调用、进程修改或实施反规避技术。在基于 Windows 的恶意软件中，这些操作严重依赖于 Windows API 调用结合 C/C++ 或 C# 功能。因此，鉴于原生系统 API 的复杂性和上下文限制，使用 LLM 编辑大规模的恶意软件源代码仍然是一个重大挑战。

3.2.2 (A1) 在功能级别生成恶意软件变体的框架

为了应对挑战 C1，LLMalMorph 通过几个关键阶段运作。它首先遍历恶意软件源代码文件的抽象语法树 (AST)，以系统地提取函数主体、头文件信息和全局变量声明。随后，提取出的组件作为开源 LLM 的输入，其中精心设计的提示指导函数修改过程。最后，修改后的组件被重新整合回源代码，生成原始恶意软件的功能性变体。这种方法确保了恶意软件组件的精确提取和修改，同时在整个转换过程中保持其结构完整性，且不会使 LLM 负担过重。

3.2.3 (C2) 使用基于 LLM 的函数修改时恶意软件代码库一致性保持的挑战

恶意软件项目通常跨越多个文件，其中修改一个部分通常需要跨其他相关文件进行协调更改。鉴于当前 LLM 的能力，在保持多文件代码库一致性的同时编辑源代码，在没有人工监督的情况下可能过于容易出错，因为 LLM 在多文件修改、依赖关系解析、项目级配置以及跨大型代码库的编辑方面存在困难 [34], [35]。例如，使用指定转换重构单个函数可能需要在关联的头文件中进行更新、在整个代码库中重构和重命名其用法、或添加新头文件、链接静态库、修改编译器指令、或更改整个项目的语言配置以适应 LLM 生成的代码。虽然添加头文件或在单个文件内重命名等简单任务可以实现自动化，但更复杂的多步骤修改在很大程度上取决于 LLM 生成更改的性质和特定恶意软件项目的结构，这使得一刀切的解决方案不可行。尽管像 Copilot3 这样底层使用 LLM 的技术改进了多文件处理，但上下文限制仍然是阻碍将开源 LLM 接入整个代码库的关键障碍，该代码库能够可靠地重构互连的代码库，因此在泛化到各种恶意软件项目方面存在不足。代码生成的 LLM 幻觉问题 [36], [37] 加剧了这些限制，导致新的复杂问题，如在代码生成过程中使用虚构的函数或误用现有 API。因此，调试 LLM 通常涉及试错，且其当前能力不足以处理超出简单语法和逻辑问题的复杂错误修复。

3.2.4 (A2) 融入人在回路流程

为了解决 C2，我们选择了一种部分自动化的解决方案来生成恶意软件功能变体。如 A1 所述，我们以自动化方式生成具有功能转换的源代码。然而，为了保持一致性和正确性，我们采用人在回路流程来处理跨多文件恶意软件项目的复杂调试和配置更改。

第 4 章 LLMalMorph 的详细设计

4.1 A. LLMalMorph 框架

在本节中，我们将详细阐述我们框架的架构（见图4.1）。LLMalMorph 分为两个主要模块。第一个模块，功能变异模块使用 LLM 和策略性生成的提示来转换恶意软件源代码函数。第二个模块，变种合成模块将转换后的函数集成回源代码，编译修改后的项目以生成恶意软件变体。该模块还融入了人在回路流程用于编译期间的调试。第一个模块又包含三个关键子模块：Extractor、Prompt Generator 和 LLM Based Function Modifier。第二个模块包含两个主要子模块：Merger 以及 Compilation and Debugging。我们现在介绍支撑该框架的形式化算法，随后对各模块进行详细解释。

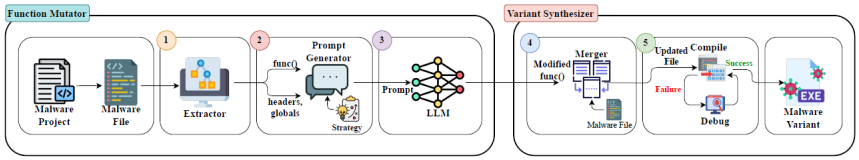

图 4.1 LLMalMorph 整体架构。该框架由两大核心模块构成：功能变异模块：从恶意软件源代码文件中提取功能函数，并借助 LLM 进行修改。变种合成模块：将修改后的函数更新至恶意软件源代码，通过编译项目生成变种文件

Algorithm 1，专为功能变异模块设计，详细说明了三个子模块 Extractor、Prompt Generator 和 LLM-Based Function Modifier 如何转换恶意软件源代码中的函数。该算法以文件名 i、要修改的函数数量 j、期望的转换策略 s 以及选定的 LLM 作为输入。接下来，我们将详细描述每个子模块。

算法 1: 使用 LLM 转换函数

输入: 文件名 i，需要改变的函数数量 j，转换策略 s，语言模型 LLM
输出: 转换后的函数集 $\hat{F}_s = \{\hat{f}_1^i, \hat{f}_2^i ..., \hat{f}_j^i\}$

Headers,globals,functions$\{f_1^i, f_2^i, ..., f_G^i\} \leftarrow extractor(i)$;
初始化转换后的函数集 $\hat{F}_s \leftarrow \emptyset$;
for $t \leftarrow 1$ **to** j **do**
    $p_s || f_t^i \leftarrow gen\_prompt(s, f_t^i, headers, globals)$;
    Transform function: $\hat{f}_t^i \leftarrow LLM(p_s || f_t^i)$;
    Update set: $\hat{F}_s \leftarrow \hat{F}_s \cup \{\hat{f}_t^i\}$;
return $\hat{F}_s$

4.1.1 Extractor 子模块

Extractor 子模块（算法 1 的第 1 行）利用 extractor 子程序，该子程序接收一个源文件并遍历源代码的解析树。它从解析树中提取并存储以下两条辅助信息：全局声明的变量、结构体、编译器指令的列表，并将它们存储于 $globals$ 中；以及所包含头文件的列表，并将它们存储于 $headers$ 中。此类信息对于成功转换至关重要，因为它提供了函数可能使用的全局依赖项的基本上下文。以提示的形式将此上下文提供给 LLM 可确保生成更准确且语法正确的代码。此后，该子程序解析源文件以提取所有函数定义，生成集合 $\{f_1^i, f_2^i, ..., f_G^i\}$。

4.1.2 Prompt Generator 子模块

算法 1 的第 3−7 行对应于 Prompt Generator 和 LLM Transformation 子模块。第 4 行中的子程序 gen_prompt 被调用，参数为函数 f_t^i、转换策略 s 以及提取的 $headers$ 和 $globals$。它将输入代码和策略构造成一个为 LLM 定制的提示 $p_s||f_t^i$。提示的设计详见第 IV-C 节。另请参阅附录 F 了解该子程序中使用的不同类型的提示，以及附录 G 了解一个完整构建的提示及其相应 LLM 响应的示例。

4.1.3 LLM Based Function Modifier 子模块

算法 1的第 5 行将设计好的提示 $p_s||f_t^i$ 提供给选定的 LLM，并获取转换后的函数。在代码生成过程中，我们使用了 LLM 的默认推理设置。具体而言，temperature=0.8，top-k=40，top-p=0.9。我们在附录 A-A 中提供了使用 LLM 进行代码生成过程的详细描述。

最后，第 6 行将转换后的函数 f_t^i 追加到输出集合中。一旦所有选定的函数处理完毕，该算法即返回转换后的集合。我们注意到，该算法可以执行多次，以从同一源文件生成函数的多个变体。然而，在本工作中，对于每个选定的恶意软件样本，我们将评估限制在转换后函数的单一版本上。

Algorithm 2，在变种合成模块中实现，使用由算法 1 产生的转换后函数集合 $\hat{F}_s$、恶意软件项目 P 以及被修改的文件 i。它以增量方式生成恶意软件变体，并结合手动调试以确保成功编译。第 1 行初始化恶意软件变体的结果集合 M_s。该集合包含针对文件 i 使用策略 s 生成的恶意软件变体。尽管我们展示了针对某个特定文件的算法，但当我们处理后续文件时，先前处理过的文件的所有修改都会被保留并向前传递，从

而确保恶意软件代码库的累积式转换。该算法的核心功能封装在第 2-10 行中，其中每个转换后的函数被迭代地集成和调试。

算法 2: 恶意软件变种生成

输入: 恶意软件项目 P，文件名 i，转换后的函数集合 $\{\hat{f}_1^i, \hat{f}_2^i ..., \hat{f}_j^i\}$
输出: 编译后的恶意软件变种集合 M_s

1 初始化集合: $M_s \leftarrow \emptyset$;
2 **for** $t \leftarrow 1$ **to** j **do**
3 Extract subset of functions: $\hat{F}_t \leftarrow \{\hat{f}_k^i \in \hat{F}_s | 1 \leq k \leq t\}$;
4 Generate updated file: $\hat{i} \leftarrow merger(i, \hat{F}_t)$;
5 **while** $compile(P)$ *fails* **do**
6 Debug project P and resolve errors;
7 Compile project: $\hat{M}_s \leftarrow compile(P)$;
8 Add compiled malware: $M_s \leftarrow M_s \cup \hat{M}_s$;
9 return M_s

结论

参考文献

攻读学位期间发表论文与研究成果清单

致谢